Tutorial: Optimizing Applications for Performance on POWER

Application Optimization using Parallelism

SCICOMP13

July 2007

Agenda

- Shared Memory
  - Computer Architecture
  - Programming Paradigm
- pThreads
- OpenMP Compilers
  - Implementation
  - Compiler Internals
- Automatic Parallelization
- Work Distribution
- Performance
- Problems and Concerns

Parallel programming is essential to exploit modern computer architectures

- Single processor performance is reaching limits
  - Moore's Law still holds for transistor density, but…
  - Frequency is limited by heat dissipation and signal cross talk
  - Multi-core chips are everywhere...
- Advances in network technology allow for extreme parallelization

Parallel choices

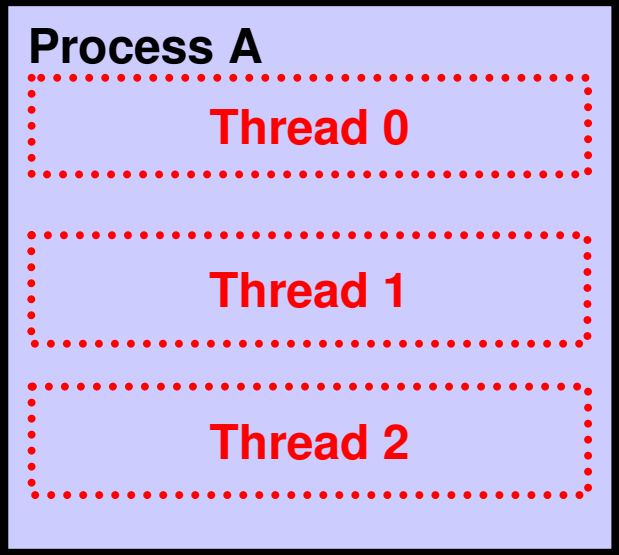
- MPI
  - Good for tightly coupled computations
  - Exploits all networks and all OS
  - No limit on number of processors
  - Significant programming effort; debugging can be difficult
  - Master/Slave paradigm is supported, as well
- OpenMP
  - Easy to get parallel speed up
  - Limited to SMP (single OS image)
  - Typically applied at loop level ← limited scalability
- Hybrid
  - OpenMP under MPI
  - Less common, but intermediate programming effort
- Automatic parallelization by compiler
  - Need clean programming to get advantage
  - Performance penalty if “bad” loops parallelized
- pthreads = Posix threads
  - Good for loosely coupled computations
  - User controlled instantiation and locks (complicated)
- fork/execl
  - Standard Unix/Linux technique

Parallel programming recommendations (for scientific and engineering computations)

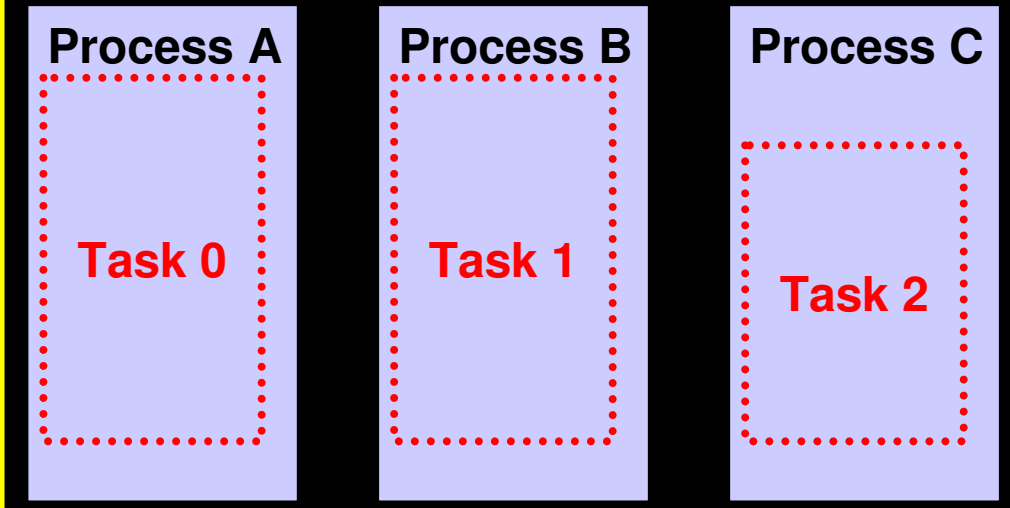
- Use MPI if possible
  - Performance on SMP node is almost always at least as good as OpenMP
  - For 1-D, 2-D domain decomposition: schedule 1-2 months work
  - For 3-D domain decomposition: schedule 3-4 months
- OpenMP can get some parallel speed up with minimal effort
  - 1 week to get 60% efficient on 8 CPUs; 3 weeks to get 80%
  - May get best performance with -qsmp=omp instead of relying on compiler to auto-parallelize for older codes
    - Can use -qsmp -qreport=smplist to get candidate loops.
- Hybrid is also possible
  - OpenMP under MPI
- pthreads are fine. Use them if it makes sense for your program.

Shared Memory Programming vs. Distributed Memory Programming

- Shared memory Single process ID for all threads
  - List threads
    - ps –m -oTHREAD



- Distributed memory
  - Each “task” has own process ID
  - List tasks:
    - ps



Shared Memory Programming

pThreads

- POSIX standard
- Library functions
- Explicit fork and join
- Explicit synchronization
- Explicit locks
- Often used in “operational” environment
- Often used for M:N applications = many short tasks for a few processors

OpenMP

- Industry Standard
- Compiler assist:
  - Directives (Fortran)
  - Pragmas (C, C++)
- Explicit “fork” and “join”
- Implicit synchronization
- Implicit locks
- Often used in data analysis
- Often used for 1:1 applications = one task per processor

OpenMP and pThreads

- OpenMP uses a shared memory model similar to pThreads
  - Fork
  - Join
  - Barriers (mutexes)
- NOT strictly built on top of pThreads

OpenMP and pThreads

```
Void sub(n,A,B)
{

      #pragma omp parallel
      {
      #pragma omp for
            for (i=0;i<n;i++)
                  {
                    A[i] = func(B);
                  …
                  }
      }
            ....
}
```

```
//"Master" thread forks slave threa

main()
{
...
for (i = 0; i < num_threads; i++)
      pthread_create ( &tid[i],...
for (i = 0; i < num_threads; i++)
      pthread_join( &tid[i],  … );

}

void do_work(void *it_num)
 {for (i=start;i<ending;i++)
            A[i]=func(B);
    return ;
}
```

Parallel Regions and Work Sharing

```
Void  user_sub()
{
...
#pragma  omp parallel
{
...
#pragma omp for
 for (i=0;i<;i++){
      A[i] = ....
      B[i] = ....
     }
...
}

Return (0)
}
```

- Parallel Region
  - Master thread forks slave threads
  - Slave threads enter “re-entrant” code
- Work Sharing
  - Slave threads collective divide work
  - Various scheduling schemes
    - User choice

pThreads Thread Scope

- AIXTHREAD_SCOPE={S|P}
  - S: Thread is bound to a kernel thread
    - Scheduled by the kernel.
  - P: Thread is subject to the user scheduler
    - Does not have a dedicated kernel thread
    - Sleeps in user mode
    - Placed on the user run queue when waiting for a processor
    - Subjected to time slicing by the user scheduler
    - Priority of thread is controlled by user

pThreads Thread Scope

- Typically, user is concerned with process threads
- Typically, operating system uses system threads
  - Process to system thread mapping (pThreads default):
    - 8 to 1
      - –8 process threads map onto 1 system thread
- User can choose system or process threads
  - pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);
  - export AIXTHREAD_SCOPE=S
  - export AIXTHREAD_MNRATIO=m:n

Compiling and Running an OpenMP Program

```
xlf90_r -O3 -qsmp -c openmp_prog.f
xlf90_r -qsmp -o a.out openmp_prog.o
export OMP_NUM_THREADS=4
./a.out &
ps -m -o THREAD
```

_r = reentrant code
Use -qsmp both for compiling and for linking

Compiler Transformations for OpenMP

- Outlining
  - Single framework for parallelization and worksharing constructs
  - Code is moved into a contained subroutine
  - Outlined subroutine is invoked through the SMP runtime
- Simple directives are transformed into direct calls to the SMP runtime

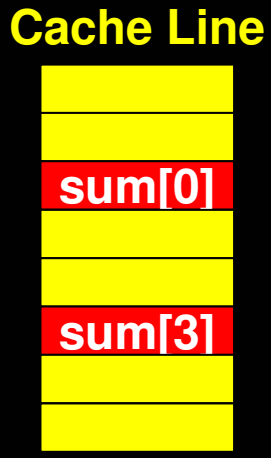
Performance Concerns

- False Sharing
  - Cache coherency thrashing
- Load balance
  - Uneven number distribution
- Barriers
  - Synchronization is expensive

False Sharing

- Multiple processors (threads) write to same cache line
  - Valid shared memory operation but causes severe performance penalty
  - Common in older Cray parallel/vector codes
- Dangerous programming practice
  - Difficult to detect

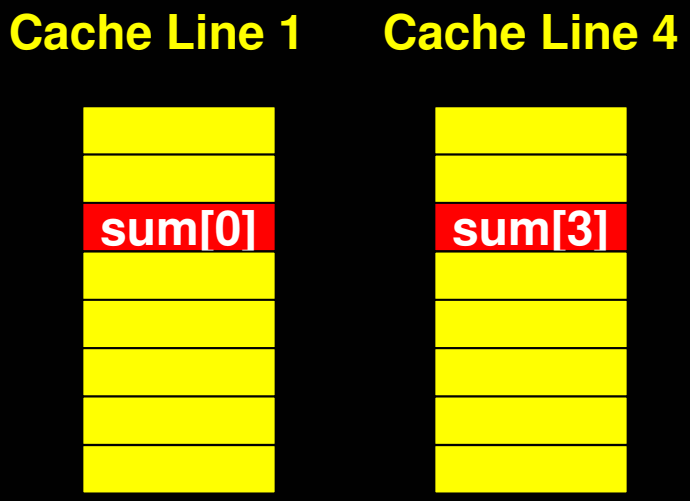
```
float sum[8];
#pragma omp parallel
p = ...my thread number...
#pragma omp for
for (i=1;i<n;i++)
   sum[p] = sum[p] + func(i,n);
```



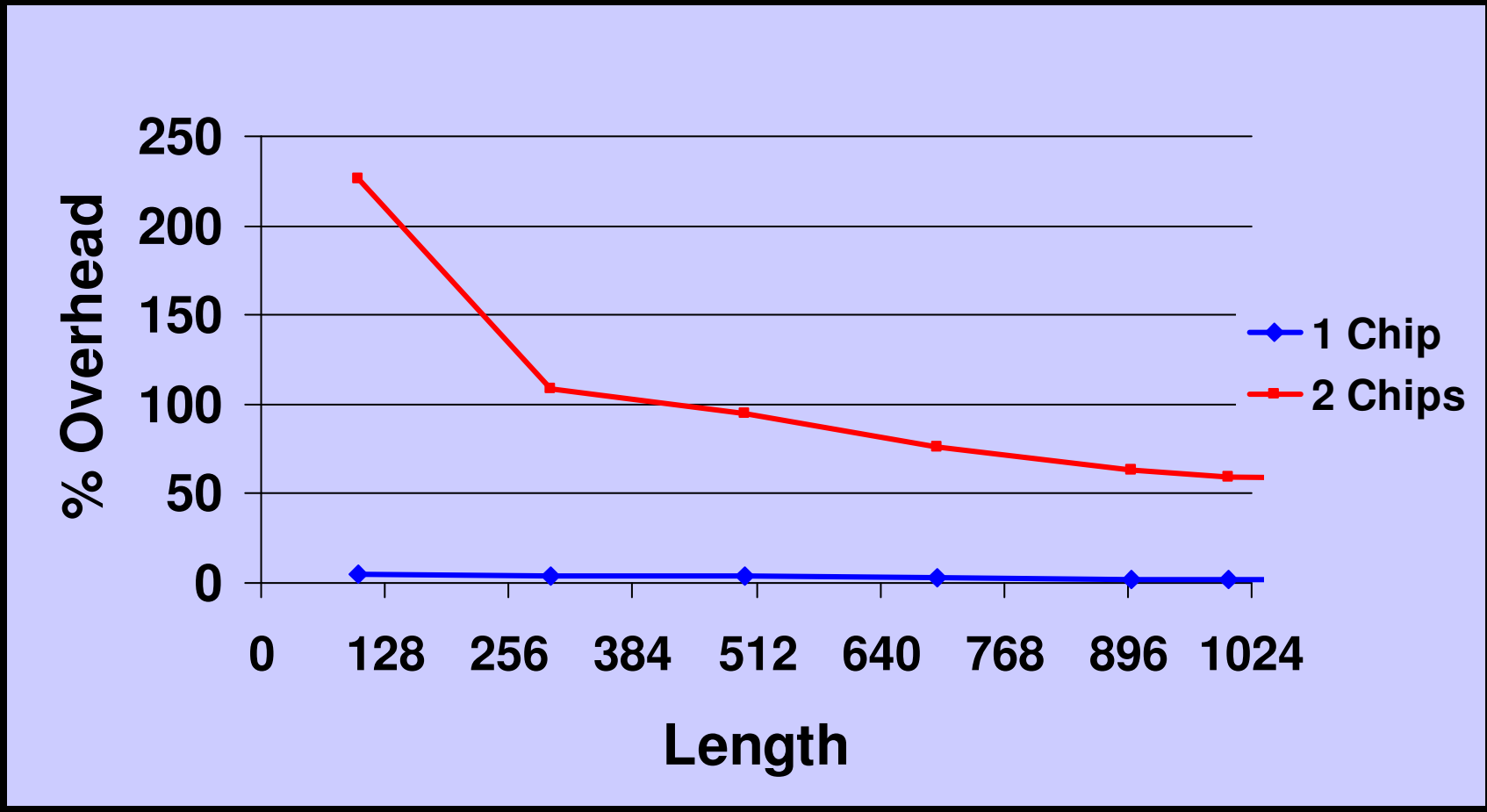
Corrected False Sharing

- Each processor (thread) writes to own cache line
  - Wastes a tiny bit of memory
  - Cache line is 128 bytes = thirty-two 4-byte words

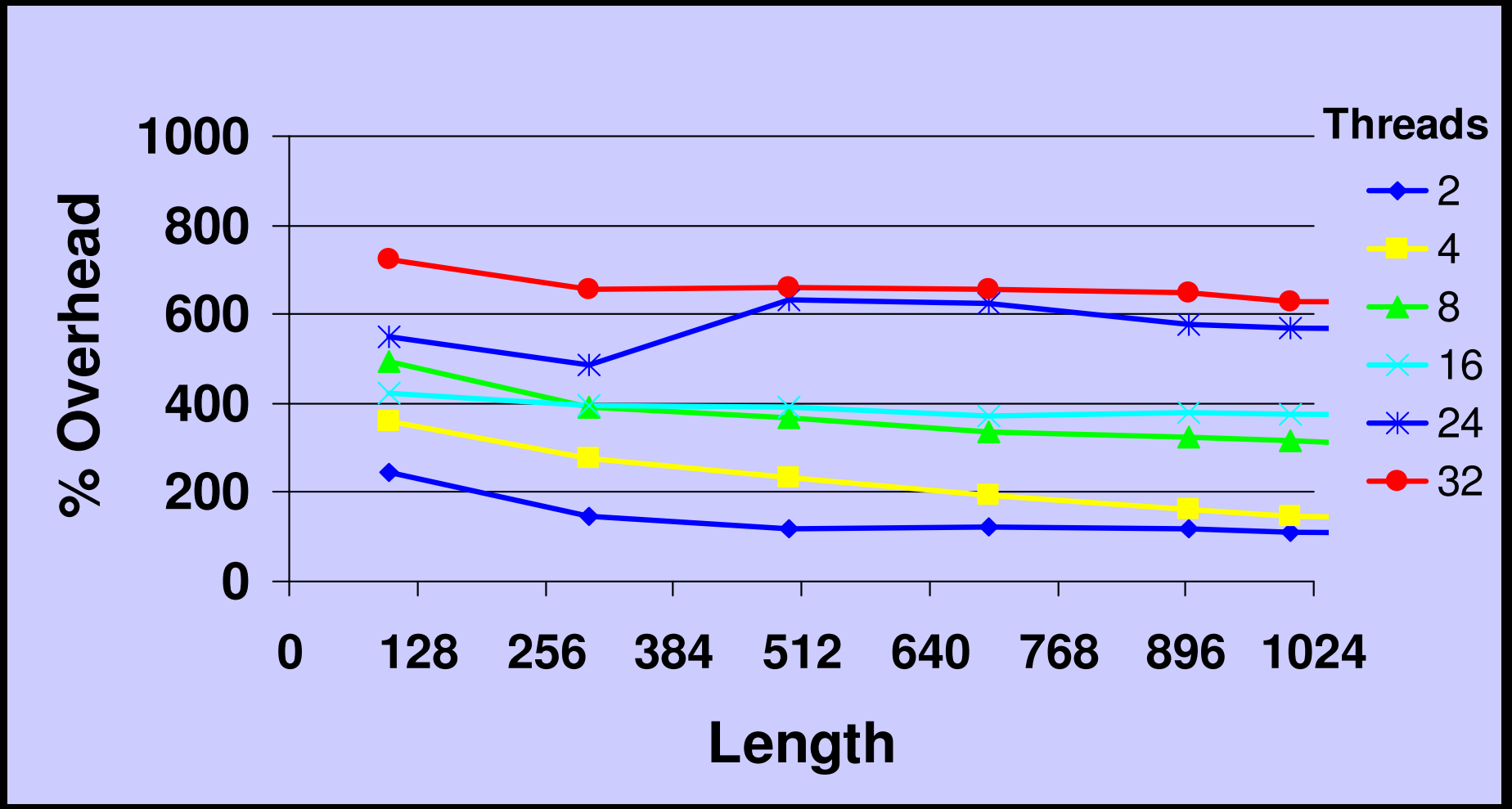
```
float sum[8*32];
int p;
#pragma omp parallel
p = omp_get_thread_num();
#pragma omp for
for (i=1;i<n;i++)
   sum[p*32] = sum[p*32] + func(i,n);
```



Effect of False Sharing: Two threads on Same Chip



Effect of False Sharing



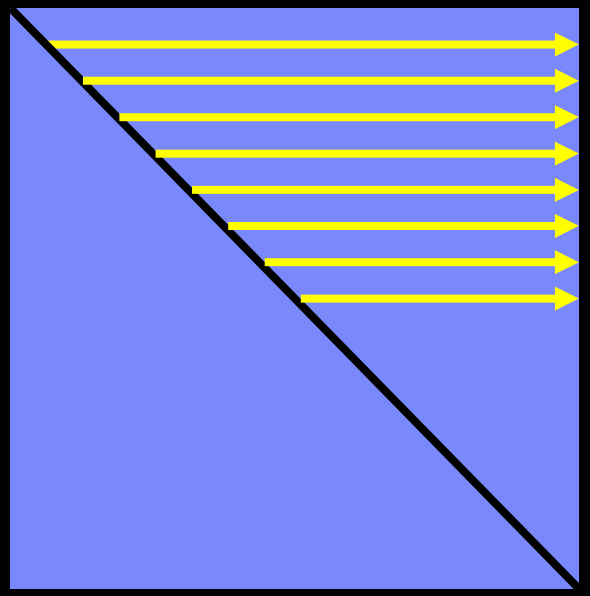
Scheduling

- Environment variable:
  - OMP_SCHEDULE={static, dynamic, guided}
- Pragma:
  - #pragma omp for schedule({static, dynamic, guided})
- Full syntax:
  - OMP_SCHEDULE={s…,d…,g…}[(chunk_size)]
  - #pragma … schedule({s…,d…,g…},[chunk_size])

Load Balance and Scheduling

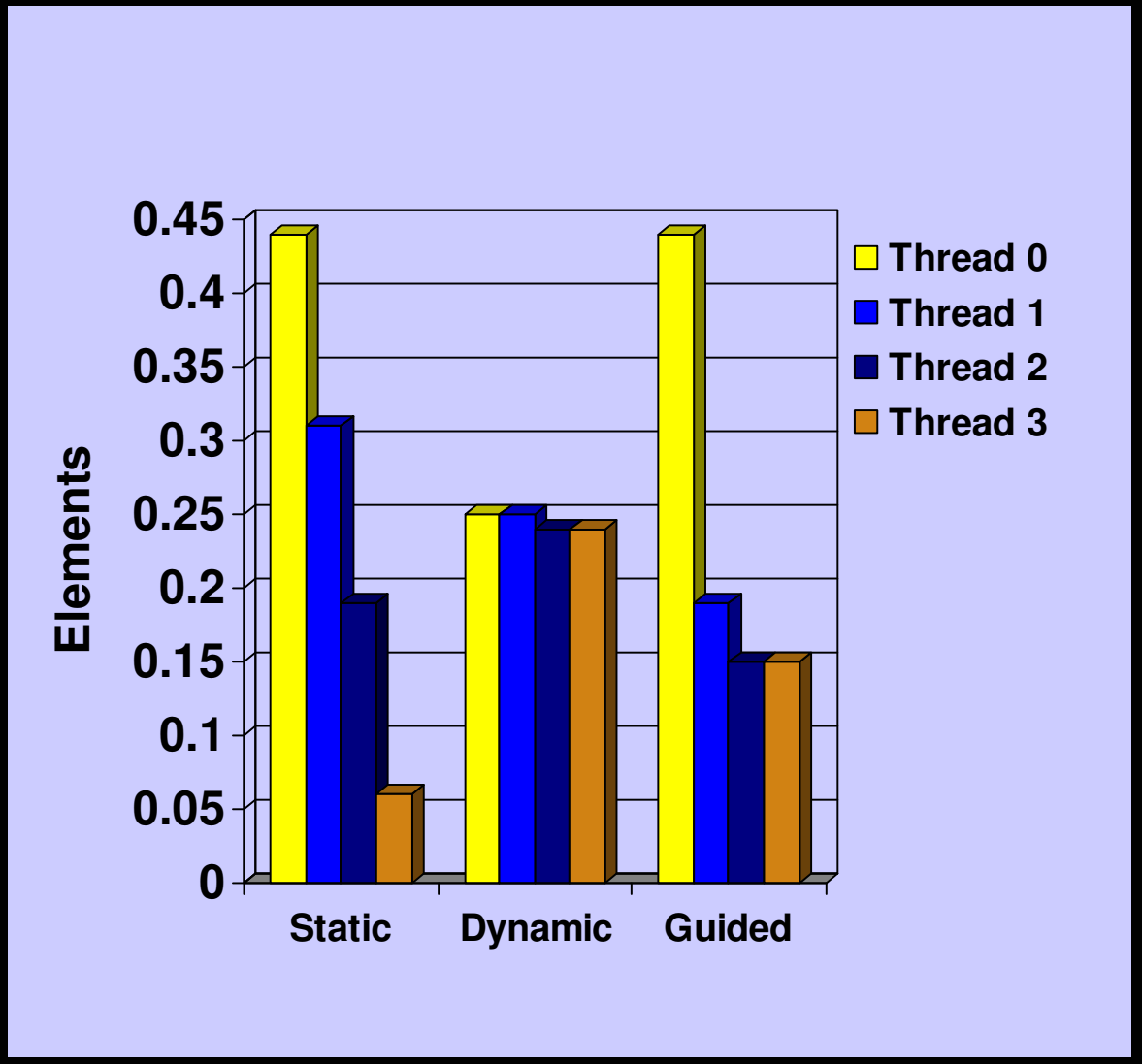
- Unbalanced constructs can be handled dynamically with scheduling
- Forward triangle example:

```
!$omp parallel do private(k,j)
do k = 1,n
        do j = k,n
          D(j,k)=D(j,k)+F(k)*F(j)
        end do
 end do
```



Forward Triangle Workload Distribution

- Static:
  - First thread gets 7/16 of the work
  - Second thread get 5/16 of the work
  - …
- Dynamic:
  - Better distribution if slab size is smaller
- Guided:
  - Very bad again



nowait

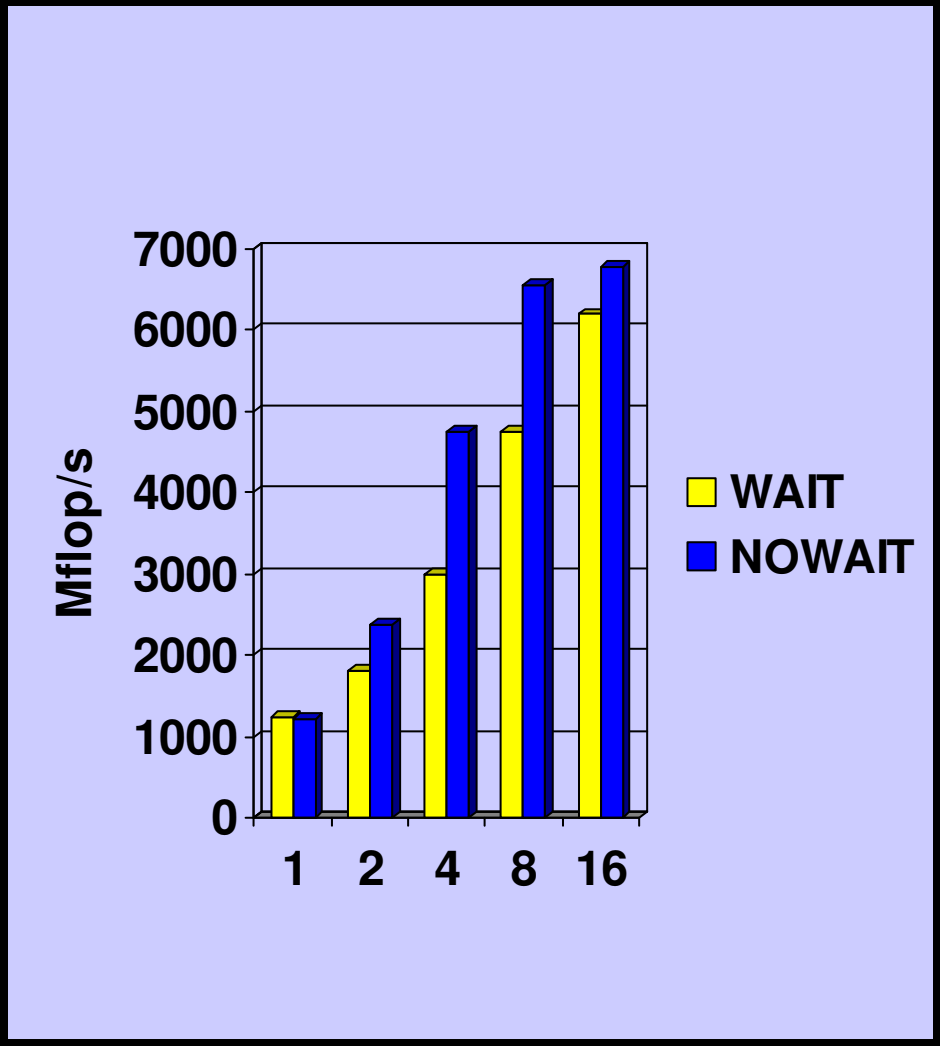
- Work sharing contracts (“omp for”) have implicit barrier at end
  - Expensive
  - Not always necessary
- #pragma omp for nowait
  - Remove implied barrier

Example

```
#pragma omp parallel private(j)
{
#pragma omp for  nowait
      for (i=0;i<n;i++)
         {….A, B, C…
          }
     ….
#pragma omp for nowait
       for (i=0;i<n;i++)
          { … D, E, F …
           }
}
```

Effect of nowait

- NOT ALWAYS VALID
  - Dependencies
- Performance always better results with no wait



Critical Regions

- Protect “one-at-at-time” sections:

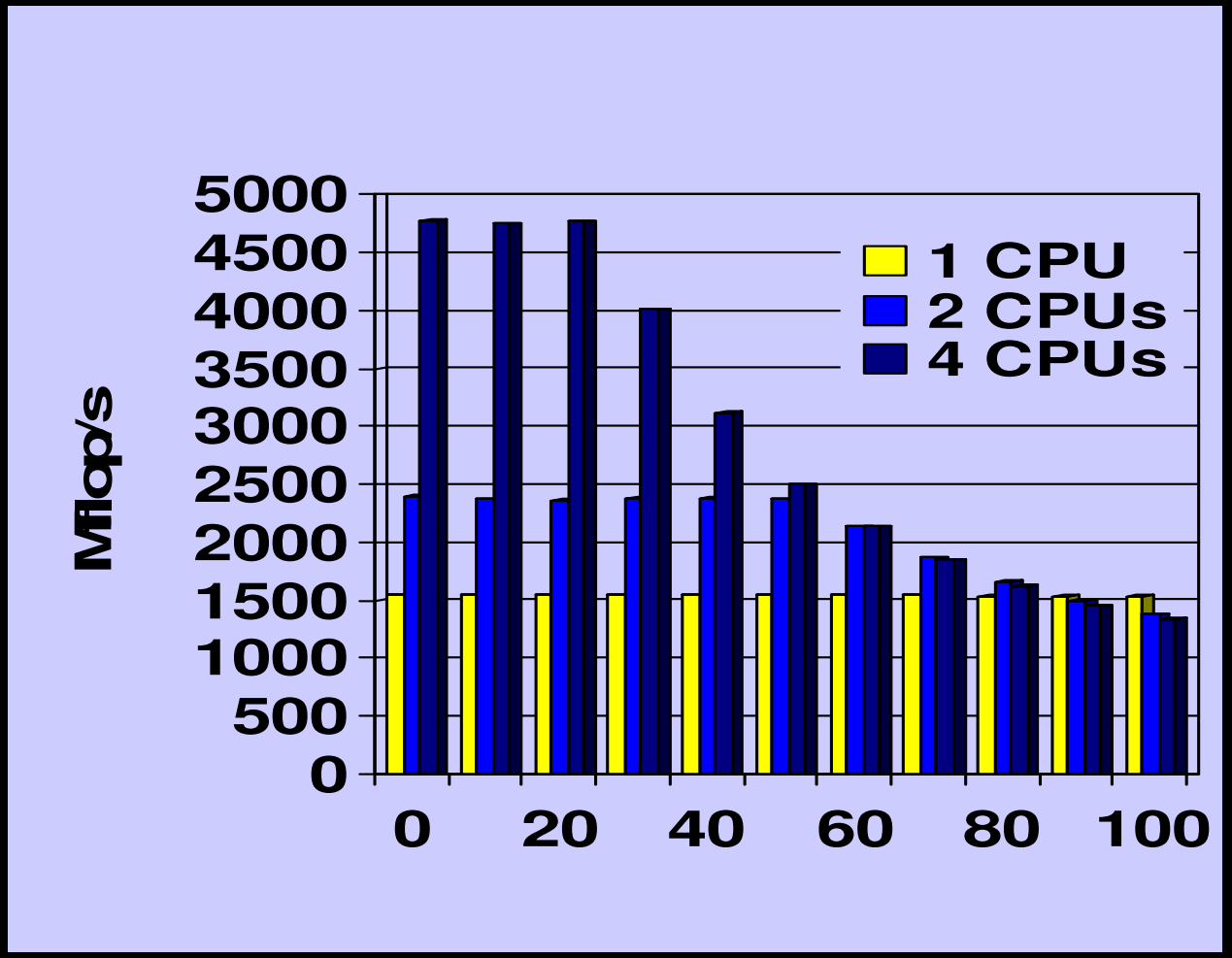
```
#pragma omp parallel
{
#pragma omp critical (x_axis)
      function_X(j);

Other_work();

 #pragma omp critical (y_axis)
      function_Y(j);
}
```

Effect of Critical Region Size

- Effect more dramatic with more threads
- Approaches single thread performance at 100% critical



Environmental Variables:
SPINLOOPTIME & YIELDLOOPTIME

- SPINLOOPTIME:
  - Number of times to retry a busy lock before yielding to another processor
  - Default:
    - 1 on uniprocessors
    - 40 on multiprocessors
  - It costs more to obtain a kernel thread from the system if it has been “yielded” or it is “sleeping”
  - If threads are going to sleep often (lot of idle time), then the default SPINLOOPTIME may not be high enough
  - Doesn’t hurt elapsed time to spin if you have whole node, but CPU time may be measuring useless work
  - Many benchmarks use very large values (100000 or more)
- YIELDLOOPTIME
  - Number of times to yield the processor when trying to acquire a busy lock, but remain in a runnable state. If YIELDLOOPTIME is exhausted the thread goes to sleep
  - Default: 0
  - If threads are going to sleep often (lot of idle time), then the default YIELDLOOPTIME may not be high enough.
  - Many benchmarks using a very large value for YIELDLOOPTIME (40000 or more)

Other environment variables

- AIXTHREAD_SCOPE=S
  - Force 1:1 task to kernel thread mapping
- OMP_DYNAMIC=false
  - Not nice on busy shared node, but more consistent results on non-shared node
- MALLOCMULTIHEAP=1
  - More than one thread can malloc, free, realloc
  - =1 uses up to 32 heaps.
- XLSMPOPTS
  - Auxiliary to OMP environment variables
  - If conflict, OMP variable wins

Using Automatic Parallelization in xlf and xlc

- Enabled by default under –qsmp or -qsmp=omp:auto
- Compatible with OpenMP
- Assisted by assertions in the code
  - !IBM$ INDEPENDENT
- Additional environment variables
- Runtime profiling
- XLSMPOPTS=profilefreq=n
- Loop-level automatic parallelization
  - DO
  - forall
  - for (C)
  - Array language expressions
- Data management is performed automatically by the compiler
- USE WITH DISGRESSION!

Automatic Parallelization in XLF

- Compatible with OpenMP
- Enabled with -qsmp or -qsmp=omp:auto
- Loop-level automatic parallelization
  - DO
  - forall
  - for (C)
  - Array language expressions
- Data management is performed automatically by the compiler

MPI options

- IBM Parallel Environment
  - POE
  - Highly optimized for IBM processors, adapters, and networks
  - Have to purchase license
- MPICH
  - Uses TCP/IP protocol
  - Free
- LAM MPI
  - Free
- OpenMPI
  - Free, but very new.
- etc., etc.

NOTE: Remainder of lecture assumes IBM Parallel Environment is used

Parallel Environment Documentation

- http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.pe.doc/pebooks.html

Compiling and Running an Interactive MPI Program

```
mpcc_r -c mpi_prog.c
mpcc_r -o a.out mpi_prog.o
cat << EOF > host.list
  nodename0
  nodename0
  nodename1
  nodename2
EOF

# option 1
poe a.out -procs 4 -hostfile host.list

# option 2
export MP_PROCS=4
export MP_HOSTFILE=$PWD/host.list
a.out
```

Submitting a Simple LoadLeveler Batch Job

```
#!/bin/ksh
#
# @ error                 = Error
# @ output                = Output
# @ notification          = never
# @ wall_clock_limit      = 00:59:00
# @ job_type              = parallel
# @ node                  = 2
# @ tasks_per_node        = 8
# @ network.mpi       = sn_single,shared,US
# @ node_usage            = not_shared
# @ class                 = standard
# @ queue

poe date
```

Message Passing Software

- Parallel Environment (PE)
  - LoadLeveler manages jobs
    - Multiple scheduler options: LL, LSF, PBS, Torque/Maui
  - Distributed batch queuing system
  - Parallel Operating Environment (POE)
  - Message passing library (MPI)
  - Parallel debuggers
  - Parallel Operating Environment (POE)
  - Generalization of mpirun...
- Runs on:
  - pSeries systems
  - AIX workstations
  - Linux on Power (SLES 9 and RHEL 4)
  - Linux on System x (SLES 9 and RHEL 4)
- PATH=/usr/lpp/LoadL/full/bin

Invoking MPI via Compiler

- MPI usage: prefix 'mp' before the compiler
  - mpxlf ....
  - mpcc
  - mpCC
- "mp" scripts do all the "-I" and "-L" for you
  - Sets include path
  - Links libmpi.a

Language	Compiler
Fortran 77	mpxlf
Fortran 90	mpxlf90
C	mpcc
C++	mpCC

Parallel Operating Environment (POE)

- “poe” is equivalent to MPICH "mpirun" command
  - Also distributes local environment
  - Local environment variables are exported to other nodes
- Example:
  - $ poe a.out -procs ...
    - or
  - $ a.out -procs ... # /usr/bin/poe is implied
  - $ poe ksh myscript.ksh ...
  - Runs "myscript.ksh" on nodes listed in host.list

Compile and Run and MPI program

```
$ mpcc mpiprogram.c
  # Edit host.list or batch queue script
$cat host.list
    r36n11.pbm.ihost.com
    r36n11.pbm.ihost.com
    r36n11.pbm.ihost.com
    r36n11.pbm.ihost.com
$ poe a.out –procs 4 –hostfile host.list
    Or
$ a.out –procs 4 –hostfile host.list
    Or
$ env MP_PROCS=4 MP_HOSTFILE=host.list a.out
```

Specifying MPI Control Parameters

- Interactive
  - poe a.out –procs 2 –hostfile myhost.list
- LoadLeveler (or similar LSF parameters) two out of three following inputs:
  - #@ node=4
  - #@ tasks_per_node=8
  - #@ total_tasks=32
- Environment variables for interactive
  - MP_PROCS=32
  - MP_TASKS_PER_NODE=8

Control Environment Variables

Parameter	Values	Description
MP_NODES	1 to n	Number of nodes
MP_TASKS_PER_NODE	1 to m	Tasks per node
MP_PROCS	1 to m*n	Number of processes
MP_TASKS	1 to m*N	Number of processes
MP_HOSTFILE	host.list	host name for interactive use
MP_LABELIO	{yes,no}	Label I/O with task numbers

Configuration Strategy

- Be aware of node concept
  - p5 575 single core nodes have eight cores
  - p5 575 dual core nodes have 16 cores
    - Use MP_SHARED_MEMORY=yes for on-node MPI communication
- MPI configurations:
  - Nodes and tasks per node
  - Procs and tasks per node
  - Procs
  - CPUs

MPI Tasks and Processors

- MPI tasks are usually associated 1:1 with cores, but
  - System has configuration for “mpistarters”
    - Possible to specify number of MPI tasks on each processor
- User concerns:
  - Total number of MPI tasks
  - Number of nodes
  - Number of tasks per node
  - Number of tasks per processor
    - SMT concerns

Number of Tasks (processors)

- MP_PROCS=MP_NODES* MP_TASKS_PER_NODE
  - MP_PROCS : Total number of processes
  - MP_NODES : Number of nodes to use
  - MP_TASKS_PER_NODE : number of proc. per node
- Any two variables can be specified
  - MP_TASKS_PER_NODE is (usually) the number of processors per node

Environment, Statistics and Information

Env. Variable	Values	Comment
MP_PRINTENV	yes, no	Echo environment Variables
MP_STATISTICS	yes, no	Low level statistics
MP_INFOLEVEL	{0,1,2,3,4,5,6}	Information Warnings Errors

MPI Environment Variables

- MP_PRINTENV=yes
  - Job ID: MP_PARTITION
  - Tasks: MP_PROCS
  - Nodes: MP_NODES
  - Tasks per node: MP_TASKS_PER_NODE
  - Library: MP_EUILIB=us [or ip]
  - Adaptor Name
  - IP Address
  - Striping setup
  - 64-bit Mode
  - Thread Scope: AIXTHREAD_SCOPE=S
  - Shared memory MPI: MP_SHARED_MEMORY=yes
  - Memory Affinity: MEMORY_AFFINITY=MCM
  - Thread Usage: MP_SINGLE_THREAD=yes

MPI Statistics (MP_STATISTICS=print)

```
#include "mpi.h“
#include "pm_util.h"

MPI_Init(&argc, &argv);

mpc_statistics_zero();
…
MPI_Send(…)
MPI_Recv(…)
...
 mpc_statistics_write(stdout);
 MPI_Finalize();

$ MP_STATISTICS=yes a.out
```

```
Start of task (pid=410098) statistics
MPCI: sends = 108000
MPCI: sendsComplete = 178000
MPCI: sendWaitsComplete = 108000
MPCI: recvs = 108000
MPCI: recvWaitsComplete = 108000
MPCI: earlyArrivals = 2
MPCI: earlyArrivalsMatched = 2
MPCI: lateArrivals = 107998
MPCI: shoves = 100000
MPCI: pulls = 138000
MPCI: threadedLockYields = 0
MPCI: unorderedMsgs = 0
MPCI: EA buffer high water mark= 1770784
MPCI: token starvation= 0
MPCI: envelope buffer used=53424
```

MPI INFO LEVEL

```
$ MP_INFOLEVEL=6 a.out
```

```
Task    0-   1:Hostname: r36n11.pbm.ihost.com
Task    0-   1:Job ID (MP_PARTITION): 1134540874
Task    0-   1:Number of Tasks (MP_PROCS): 2
Task    0-   1:Number of Nodes (MP_NODES): NOT SET
Task    0-   1:Number of Tasks per Node (MP_TASKS_PER_NODE): NOT SET
Task    0-   1:64 Bit Mode: YES
Task    0-   1:Threaded Library: YES
Task    0-   1:Polling Interval (MP_POLLING_INTERVAL/sec): 0.400000
Task    0-   1:Buffer Memory (MP_BUFFER_MEM/Bytes): 2800000
Task    0-   1:Max. Buffer Memory (MP_BUFFER_MEM_MAX/Bytes): 2800000
Task    0-   1:Message Eager Limit (MP_EAGER_LIMIT/Bytes): 32768

D3<L4>: Message type 20 from source 0

D1<L4>: All remote tasks have exited: maxx_errcode = 0
```

Tuning Environment Variables

Parameter	Values	Description
MP_BUFFER_MEM	0 – 64,000,000	Buffer for early arrivals
MP_EAGER_LIMIT	0 - 262144	Threshold for rendezvous protocol
MP_SHARED_MEMORY	{yes,no}	Use of shared memory on node
MP_WAIT_MODE	{poll,yield,sleep}	US default: poll IP default: yield
MP_USE_BULK_XFER	yes, no	Block Transfer: message striping
MP_EUILIB	{us,ip}	Communication Method

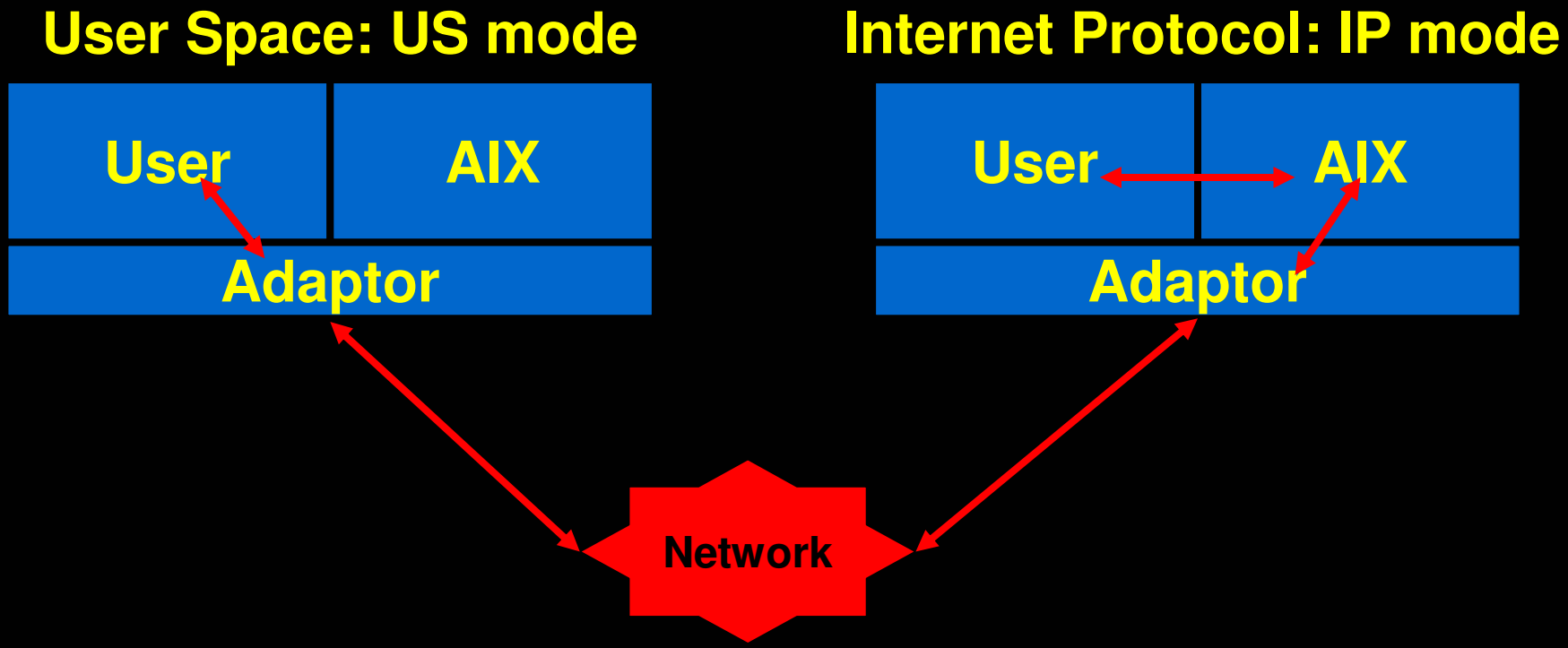
MPI Tuning

- Message Passing System Library
  - Internet Protocol (IP)
    - Ethernet protocol over Switch
  - User Space (US)
    - Low level IBM Switch protocol
- Shared Memory
  - System V shared memory for on-node messages
- Eager or Rendezvous Protocol
  - Small or large messages

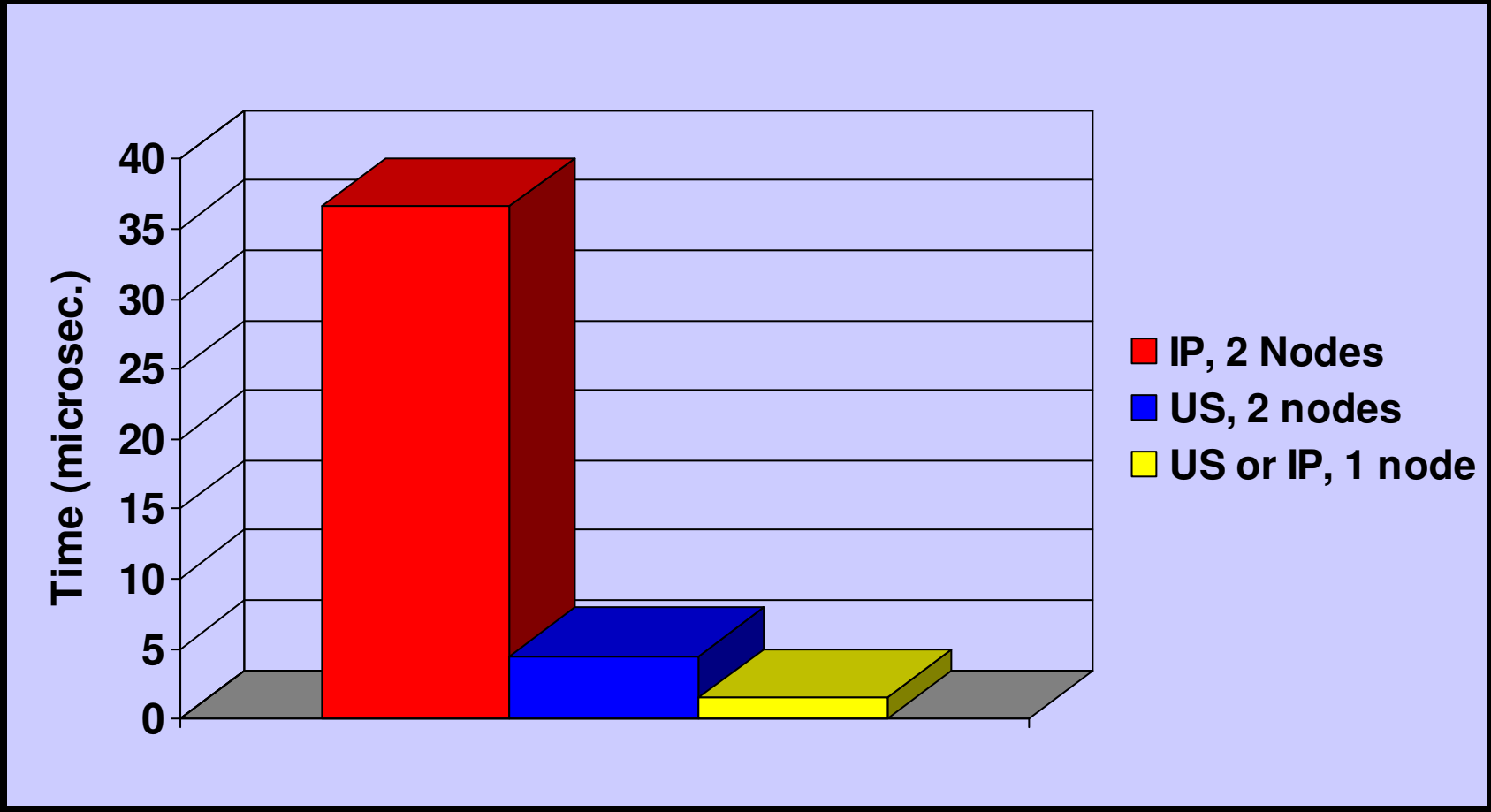
Message Passing Library

- MP_EUILIB={us,ip}
  - us: user space
    - `# @ network.mpi = sn_all,shared,US`
    - Much faster: 5 microseconds latency, 2000 Mbyte/s bandwidth
  - ip: useable with Ethernet
    - `# @ network.mpi = sn_all,shared,IP`
    - Much slower: 50 microsecond latency
- US mode is usually default

MP_EUILIB

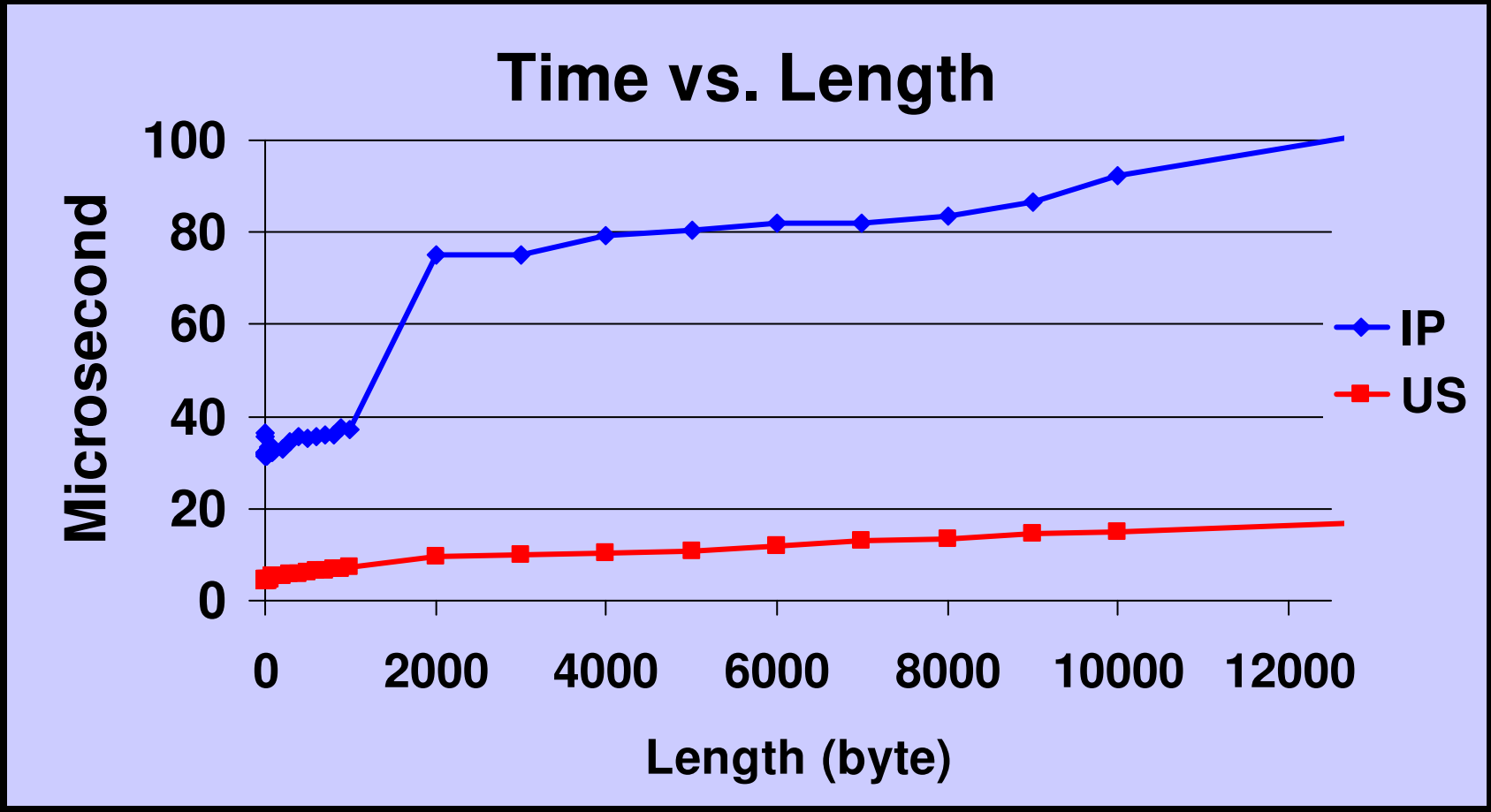


Effect of US Mode: Latency



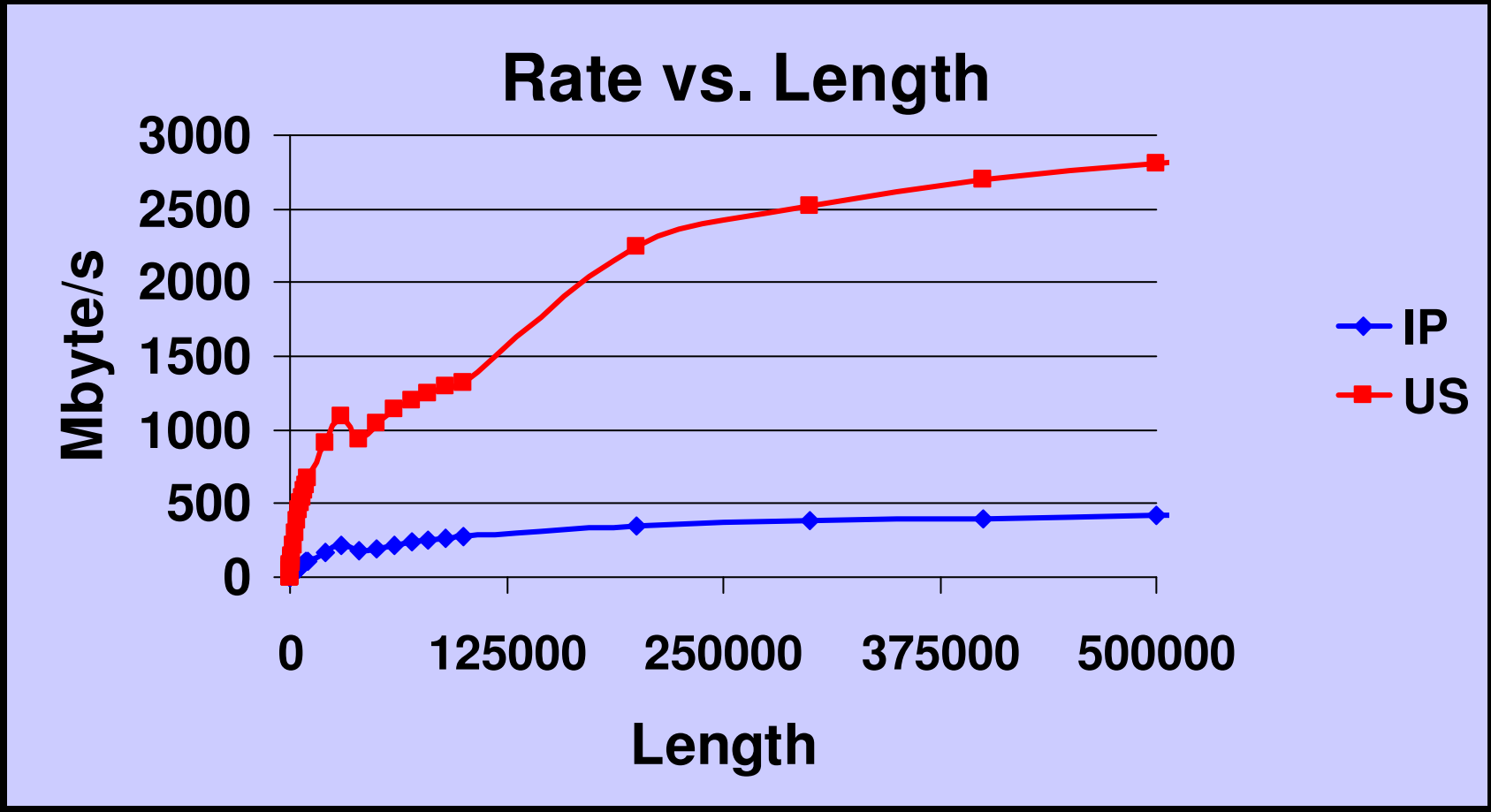
HPS Switch

Effect of US Mode: Latency



p5-575 1.9 GHZ, HPS

Effect of US Mode: Bandwidth

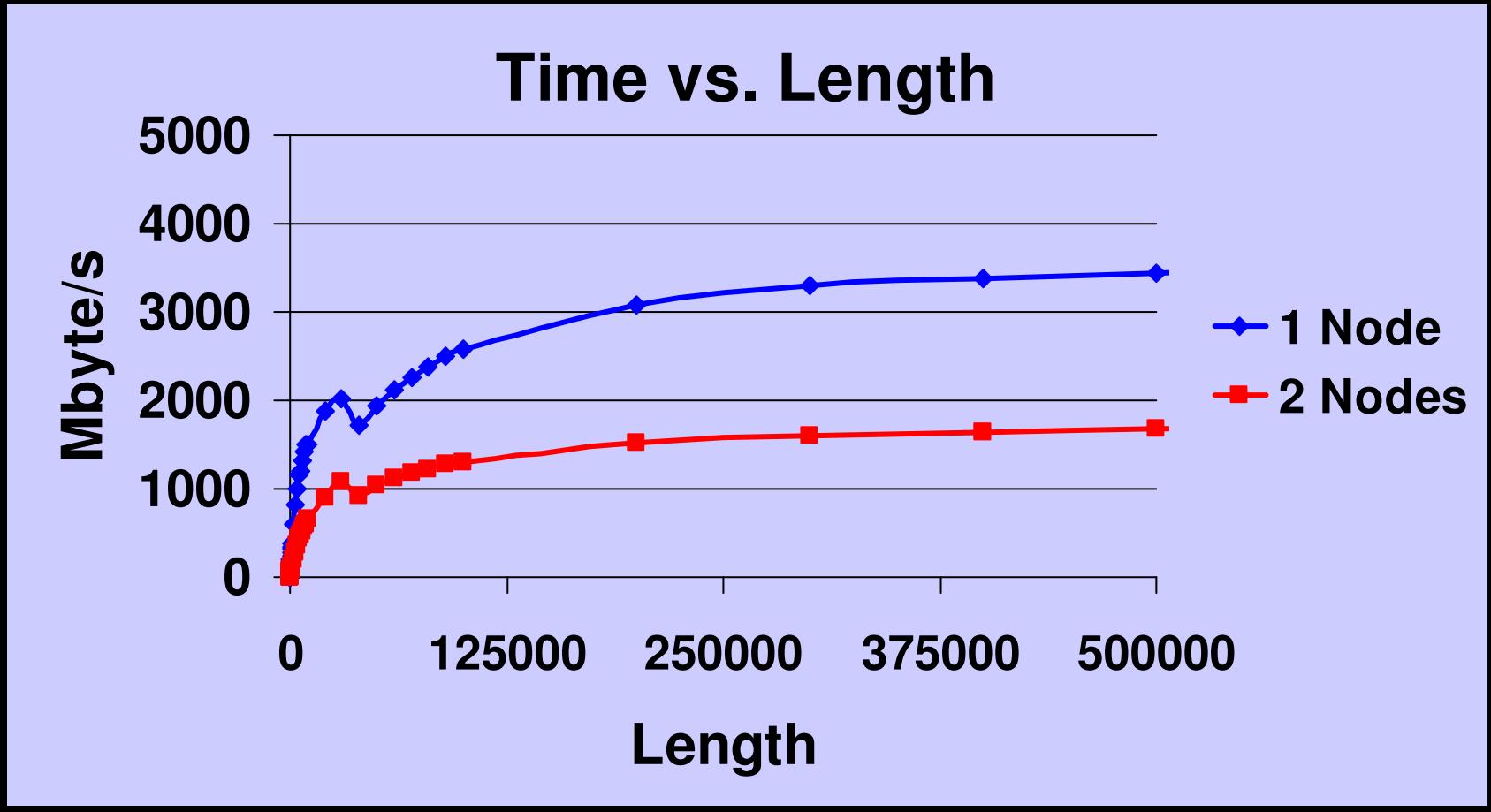


POWER5 1.9 GHZ

Shared Memory MPI

- MP_SHARED_MEMORY={yes,no}
  - High Bandwidth:
    - 2000 Gbyte/s
  - Low Latency:
    - 2 microsecond within same node
- Is now default
  - SHOULD ALWAYS BE SET TO yes (unless job would page to swap device otherwise)

MPI Performance on HPS



POWER5 1.9 GHZ

Other MPI Tuning

Env. Variable	Values	Comment
MP_SINGLE_THREAD	yes, no	1 microsec. Lower latency for non-threaded MPI tasks

NOTE: If in fact one has multiple threads, you can produce race conditions with this option. Use with caution.

Eager versus Rendevous Protocol

- MP_EAGER_LIMIT=[0-2560000000]
  - Smaller messages are passed directly to other task
  - Larger messages use rendezvous protocol

Default Eager Limits

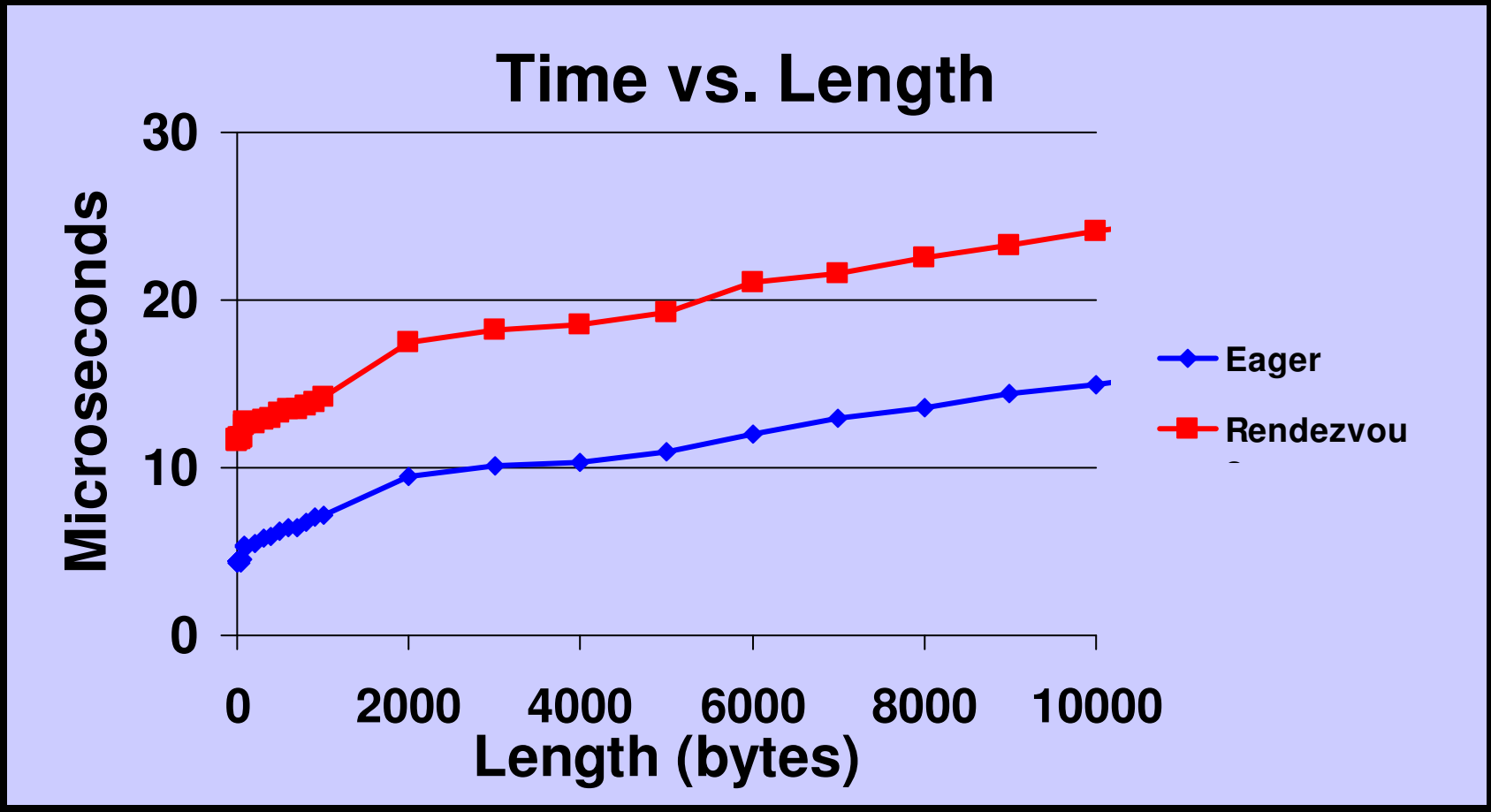
- Small Message (MP_EAGER_LIMIT)
  - Send header and message
- Large Message
  - Send header
  - Acknowledge
  - Send message

No. Tasks	MP_EAGER_LIMIT (default, bytes)
1 - 256	32768
257 – 512	16384
513 – 1024	8192
1025 – 2048	4096
2049 – 4096	2048
4097 - 8192	1024

Setting Eager Limits

- MPI library checks eager limits:
  - MP_BUFFER_MEM=2^26 (~64 Mbyte) (default)
  - MP_EAGER_LIMIT={default from table}
  - Calculate Credits:
    - MP_BUFFER_MEM / (MP_PROCS * MAX(MP_EAGER_LIMIT, 64))
  - Credits must be greater than or equal to 2.
    - MPI reduces MP_EAGER_LIMIT or increases MP_BUFFER_MEM

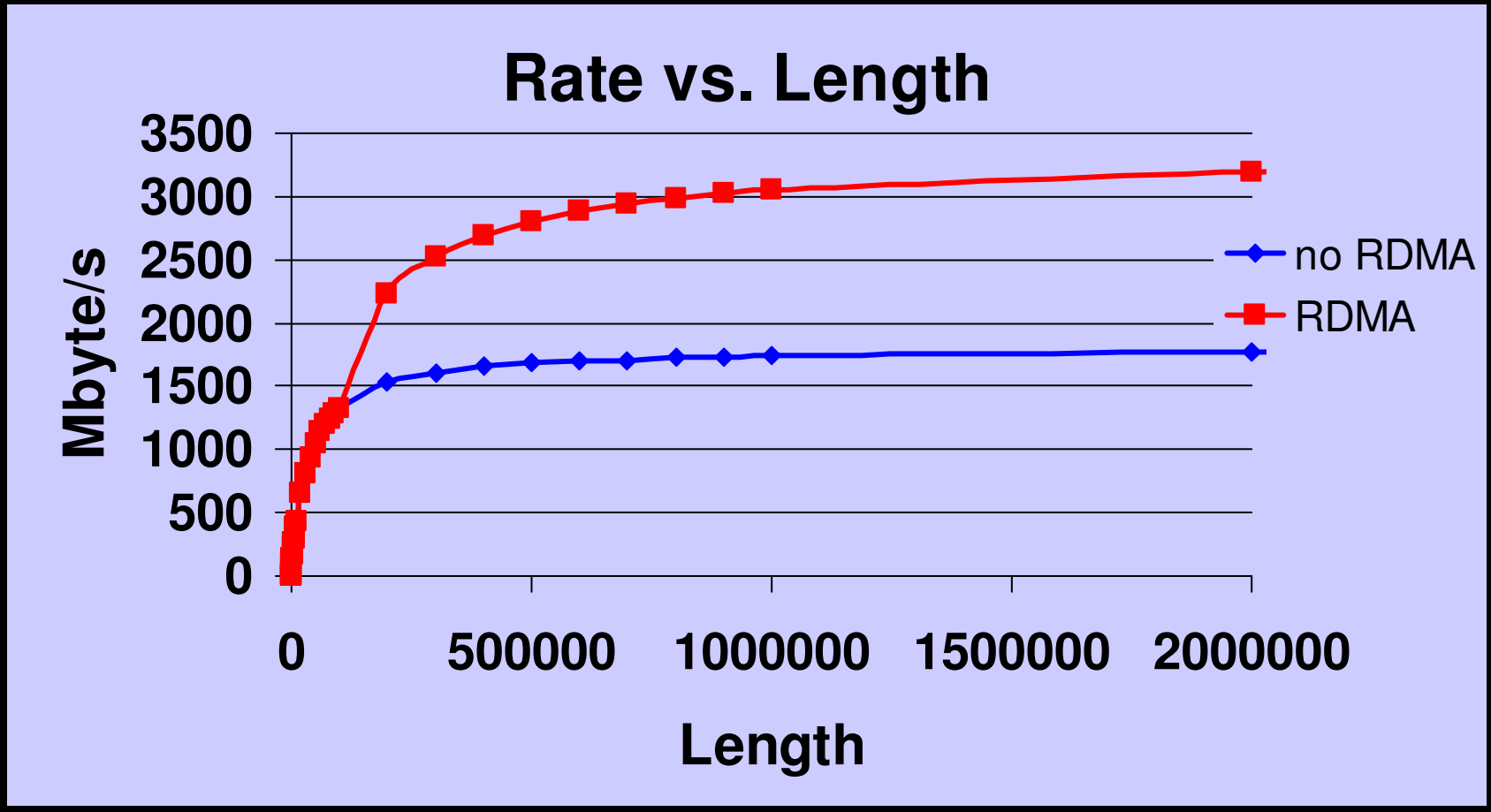
Eager vs. Rendezvous -- short messages



RDMA

- Remote Direct Memory Access
- Greatly improves large message bandwidth
  - Adapter, not CPU, copies data
  - Data backed by large or medium pages is better
- Different mechanism for interactive vs. LoadLeveler
  - LoadLeveler: #@ bulkxfer = yes
  - Interactive: MP_USE_BULK_XFER=yes

Bulk Transfer:
RDMA vs. no RDMA



P5-575 1.9 GHz, HPS, RDMA

Summary

- Shared memory programming
  - Only works on single address space node
  - Tuning:
    - Load balance and scheduling
    - False sharing
    - Critical regions
- Distrbuted memory programming
  - Performance dependant on switch speed
  - Tuning:
    - Use HPS
    - Use Shared memory MPI
    - Use EAGER protocol for small messages
    - RDMA may help